Locking vs lock-free

Contents of Lecture 11

- Purpose of using lock-free data structures
- Terminology
- Comparing locking, lock-free and transactional memory
- Lock-free data structures: stack and fifo queue

Purpose of using lock-free data structures

- Suppose we need to scale our computations to use hundreds or thousands of threads
- Two important problems with locking:
  - Limited scaling due to serialization at a lock. Severity depends on lock contention, of course.
  - Using fine grained locking *may* be complex (and lead to hard to find bugs).
- Compare speed limits in cities with traffic signals vs Autobahn.

What can we do?

Examples of ”unexpected delays”

- The thread currently owning the lock may:
    - be preempted by OS kernel due to:
        - interrupt due to disk operation completed, network packet arrived, etc
        - another thread should run
    - get a page fault (page must be fetched from disk)
    - get a TLB fault
        - translation-lookaside table fault
        - a virtual memory page translation must be updated in the CPU
        - not part of this course
    - get a cache miss

Key idea with lock-free data structures

- Use atomic variables
- Let multiple threads work on a data structure concurrently
- Detect if some other thread modified it before us
- If so, do something sensible such as update some variable and try again
- How can we detect such modifications?

Recall atomic operations

- Using assignment operators ensures an atomic read-modify-write.

```
atomic_int              a;

a += 1;
```

- The following is *not* an atomic read-modify-write.

```
atomic_int              a;

a = a + 1;
```

- We would do one atomic read, an add, and an atomic write using sequential consistency but there is no guarantee the new value is exactly one more than the old.
- For integers it is sometimes possible to use assignment operators but not always!

Another example

```
atomic_int              a;

a = f(a);
```

- For add, we can do +=
- In the general case we need something else.
- What can we do?

This is what we want to do

```
atomic_int                  a;
int                         old_a;
int                         new_a;

old_a = a;

new_a = compute_a(old_a);

a = new_a; /* but only if a == old_a */
```

- How can we do this?

Recall atomic compare exchange from Lecture 6

```
bool atomic_compare_exchange_weak(
        volatile A*     ptr,
        C*              expected,
        C               value);
```

- You can ignore the `volatile`
- Recall how it is defined:

```
    if (*ptr == *expected)
            *ptr = value;
    else
            *expected = *ptr;
```

- Operation introduced for IBM System 370
- Also called atomic compare and swap and written CAS

Using atomic compare exchange

```
atomic_int                      a;
int                             old_a;
int                             new_a;

old_a = a;
do
        new_a = compute_a(old_a);
while (!atomic_compare_exchange_weak(&a, &old_a, new_a));
```

- This modifies a only if `a == old_a`.
- If they are not equal, the current value of a is copied to `old_a`
- You may want to think of this function as:
  Is it I who should modify the variable now? (or somebody else?)
- What we essentially do is detecting a data-race and retry
- But can we be sure no other threads modified a ?

Answer to previous slide's question

- We cannot be sure.
- `a` may have been incremented and decremented back to `old_a`
- Sometimes that matters and at other times not.
- It is called the ABA-problem.
- x had value A, then B, and then A again.
- It can cause chaos if the atomic variable is e.g. a pointer to a list, and the pointer is both freed and malloced again. Then one thread may think it still has the list pointer (and can use a next field) but that will not work.
- We will come back to it later in this lecture and see a solution in detail.

Some terminology

- An algorithm is **blocking** if one thread can delay another thread. For example algorithms with mutexes are blocking.
- An algorithm is **non-blocking** if one thread cannot delay other threads.
- An algorithm is **lock-free** if at least one thread can make progress after a finite number of steps.
  This means the program makes progress but individual threads may have to wait a long time.
- An algorithm is **wait-free** if every thread can make progress after a finite number of steps.

An example: slide 9

- The code is non-blocking since there is no mutex
- Is it lock-free ?
- Or, will at least one thread leave the loop?
- Yes, the thread that was lucky to read and write the variable sufficiently close in time
- Why? Trivial if we have an atomic instruction and also true if we have load-and-reserve and store conditional, since only stores remove the reservation of another thread.
- Is it wait-free?
- Or, will every thread make progress after a finite number of iterations?
- No, an unlucky thread may loop an unbounded number of iterations

Locking vs lock-free vs transactional memory

- Locking is in some sense pessimistic
- Locking assumes there will be conflicts and avoids them
- Lock-free is optimistic
- Lock-free assumes there will be no conflict and detects them if they happen — and tries again
- Lock-free algorithms are *much* more complex to implement than blocking algorithms
- Transactional memory is also optimistic but trivial to get correct but can have performance problems when used in the wrong context.
- Which is fastest depends on the algorithm and input